



WEB TECHNOLOGIES

ReactJS

Prof. Spurthi N Anjan

Department of Computer Science and Engineering

spurthianjan@pes.edu

What is MERN Stack?



M

MongoDB

- Non-relational database(schema less)
- Stores data in JSON-like documents



E

ExpressJS

- A **server-side framework** that helps to develop the web and mobile application



R

ReactJS

- Front-end JavaScript library
- Component based
- Isomorphic: *same code can run on the server and the browser*

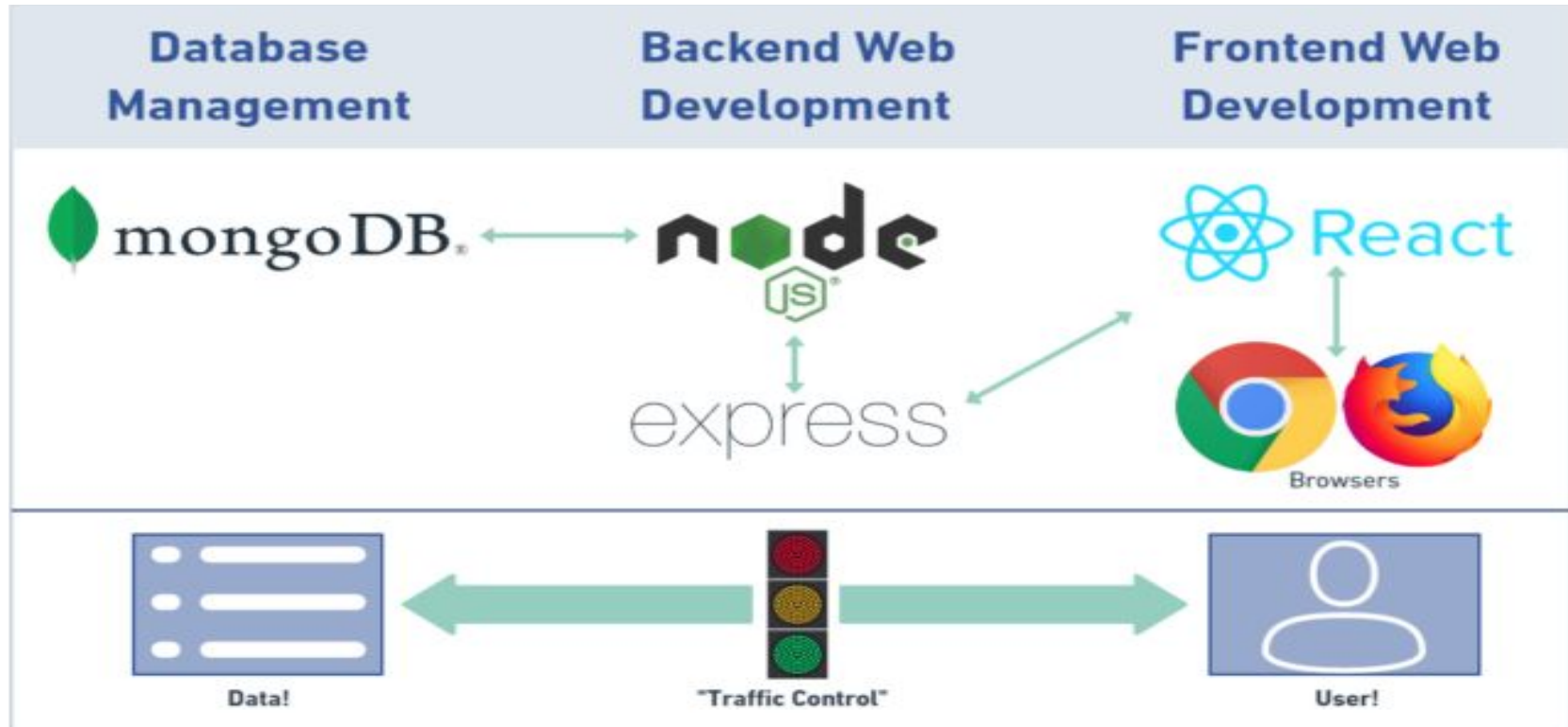


N

NodeJS

- Server-side programming framework
- Event driven
- Non-blocking I/O model.

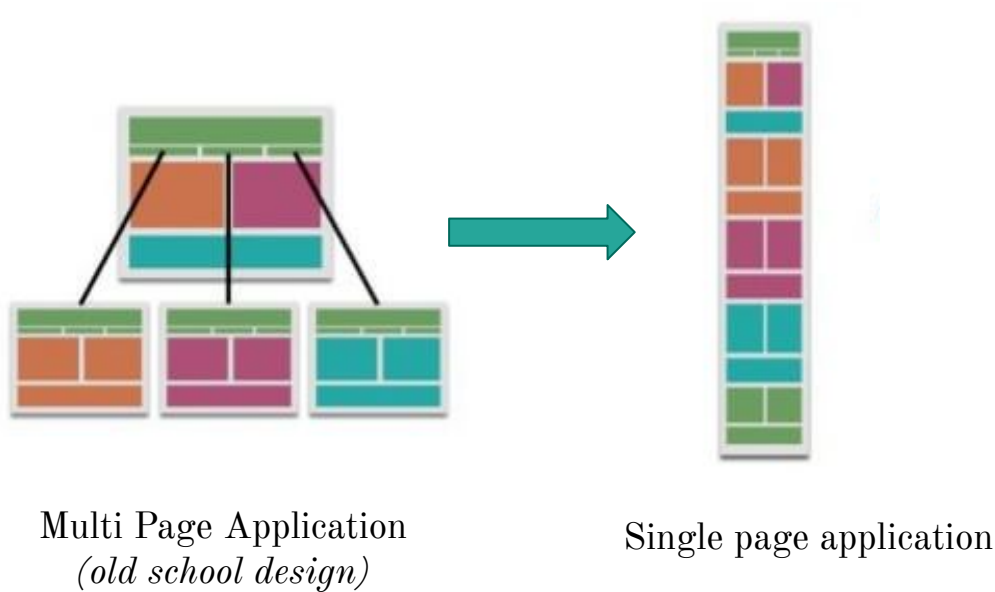
MERN Stack Development



Why MERN?

- Faster Web app development process: Easy to learn and implement technologies such as React.
- Pre-built testing tools: Easy troubleshooting.
- Full stack development: Facilitates end-to-end development
- Low learning curve: MERN is easy to learn and implement. The learning curve becomes easier if the developer is already familiar with JavaScript
- Open Source: No licensing issues with MERN
- Since MERN uses Javascript in all its tiers, developers only need to be proficient in Javascript and JSON.
- Due to its event-driven architecture and non-blocking I/O, the claim is that Node.js is very fast and a resilient web server

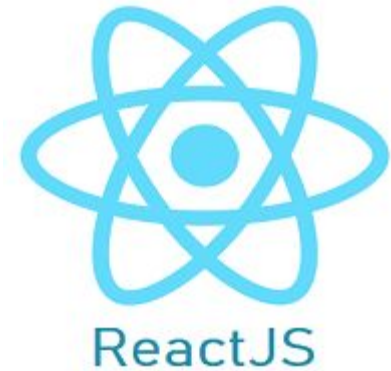
What is ReactJS?



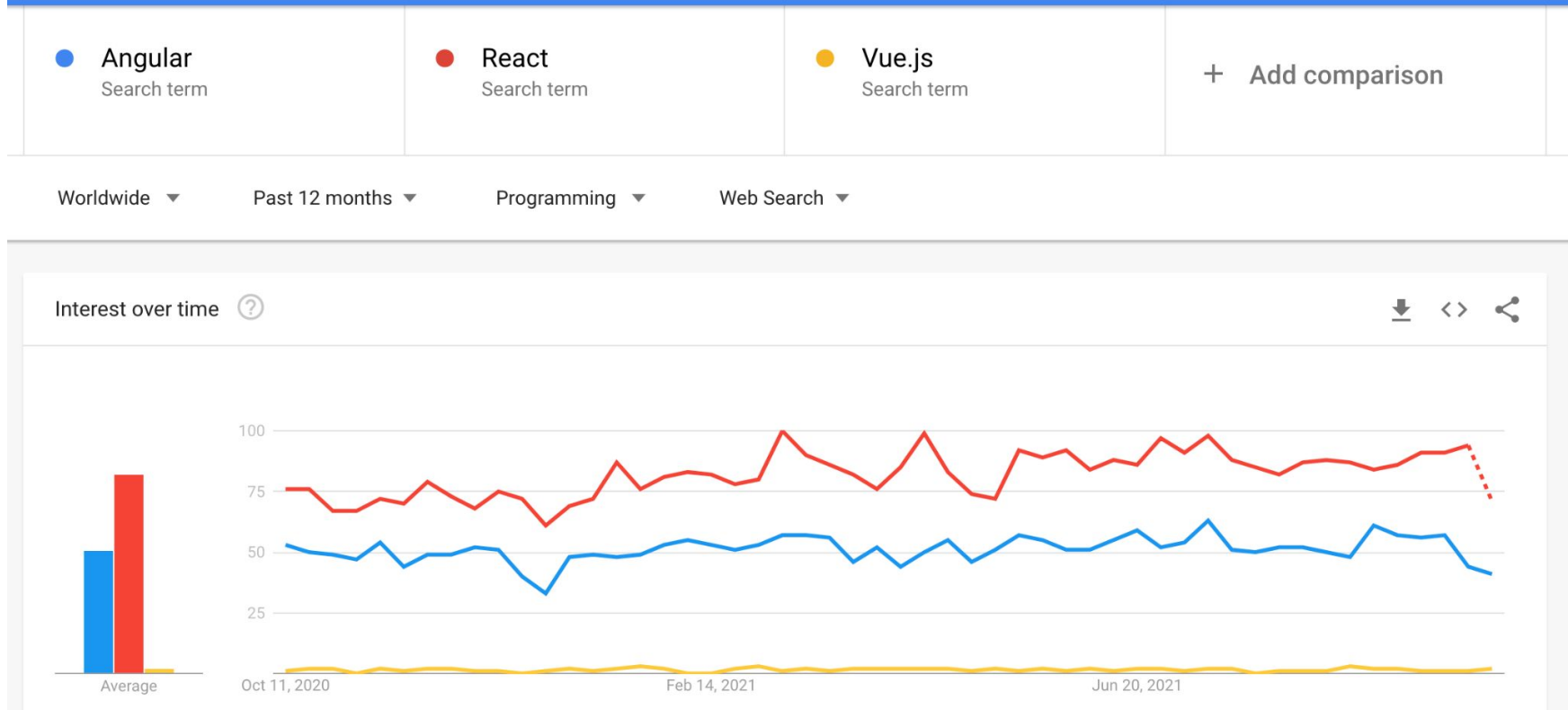
- React.js is an open-source JavaScript library that is used for building user interfaces specifically for single-page applications, developed by Facebook.
- React allows developers to create large web applications that can change data, without reloading the page.
- React creates an in-memory data structure cache which computes the changes made and then updates the browser. React library only renders components that actually change. (virtual DOM)

ReactJS Features

- Create complex UIs from small and isolated pieces of code called “components”.
- Components are the building blocks of any React application.
- Deals with the “visual” elements observed on the browser and is concerned with keeping it up-to-date
- Declarative i.e describes **what** we want instead of saying **how** to do it, as you would with imperative code.
- Facilitates fast DOM manipulation as changes are made to a virtual DOM
- Follows an unidirectional data flow; developers often nest child components within parent components.
- It can be used for the development of both web and mobile apps.
- It comes with dedicated tools for easy debugging

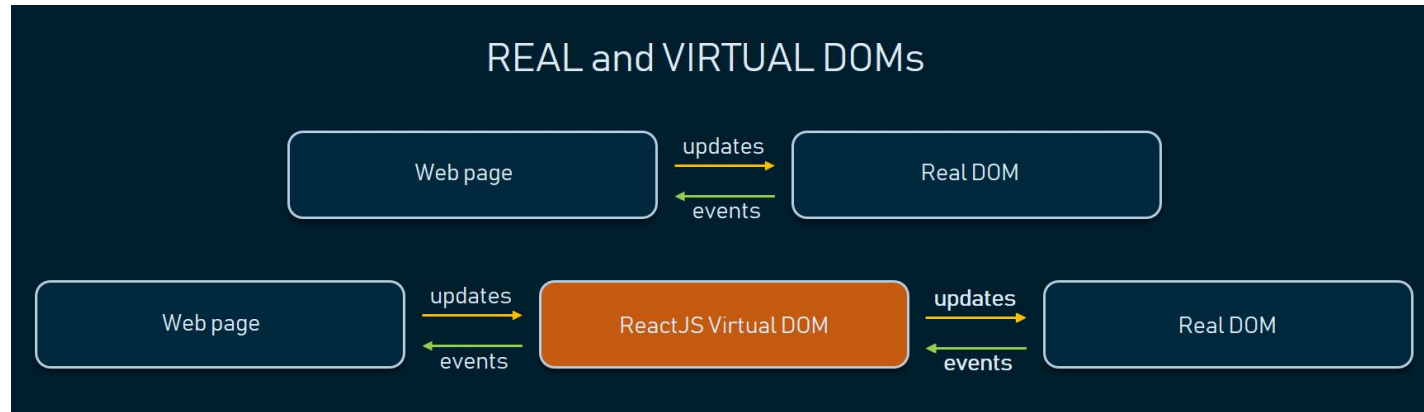


React is the popular choice!



How does React work?

- The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.
- DOM IS SLOW!
- React basically maintains a tree for you.
- React allows you to effectively reconstruct your DOM in JavaScript and push only those changes to the DOM which have actually occurred.



React Setup

Option 1: CDN

Incorporate the following lines in your script under <head>:

```
...  
<script crossorigin  
src="https://unpkg.com/react@16/umd/react.development.js"></script>  
  
<script crossorigin  
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>  
...
```

Option 2: Local Development Environment (requires the use of Node.js)

Reference: <https://reactjs.org/docs/create-a-new-react-app.html>

React Elements

- A React element is a JavaScript object that represents what is shown on screen.
- An Element contains type and properties.
- To create the React element, the createElement() function of the React module needs to be called.

Syntax: `React.createElement(type, [props], [...children])`

Type: any HTML tag such as the string 'div', or a React component

props: object containing HTML attributes or custom component properties.

The last parameter(s) is zero or more children elements, which again are created using the createElement() function itself. Since an element can be a nested collection of other elements and can depict everything on the entire screen, it is also called the virtual DOM.

Example:

```
const element = React.createElement('h1', {}, 'My First React Code')
```

Rendering Elements

- Each of these React elements needs to be transferred to the real DOM for the user interface to be constructed on screen.
- To do this, a series of `document.createElement()` calls needs to be made corresponding to each of the React elements
- The ReactDOM does this when the `ReactDOM.render()` function is called.
- This function takes in as arguments the element that needs to be rendered and the DOM element that it needs to be placed under.
- Return of `render()` has to be a single element, these elements have to be enclosed within a
- or a React Fragment component

Rendering Elements

Syntax: `ReactDOM.render(element, container[, callback])`

element: React Element to be rendered.

container: This parameter expects the container in which the element has to be rendered. Containers can also been selected using `querySelector()`, `getElementById()`, etc

callback: This is an optional parameter that expects a function that is to be executed once the render is complete.

Example:

```
ReactDOM.render(element, document.getElementById("container"));
```

A Simple React Code

```
<!DOCTYPE HTML>
<html>
<head>
<title>My first react code</title>
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
</head>
<body>
<div id="content"></div>
<script>
const element = React.createElement('div', {title: 'Outer div'}, React.createElement('h1', null, 'Hello
World!'));
ReactDOM.render(element, document.getElementById('content'));
</script>
</body>
</html>
```

Note: Hover over the text on the browser or anywhere to its right side within the boundaries of the outer div to see the tooltip “Outer div” pop up.

The Bigger Picture

The simple element that we created in the previous section was easy to write using `React.createElement()`

But imagine writing a deeply nested hierarchy of elements and components:

it can get pretty complex!

JSX to the Rescue

- React has a markup language called JSX, which stands for JavaScript XML
- JSX is a syntax extension to JavaScript
- You can use any **valid JavaScript expression**.
- It allows you to easily mix JavaScript and HTML-like tags to define user interface (UI) elements and their functionality. So our hello world element would look like so:

```
const element = (  
  <div title="Outer div">  
    <h1>Hello World!</h1>  
  </div>a  
) ;
```

- React doesn't require using JSX but it surely helps and is very recommended !

Using JSX

- Browsers' JavaScript engines don't understand JSX
- It has to be transformed into regular JavaScript based `React.createElement()` calls
- A compiler is needed: Babel
- The reference to the Babel JavaScript compiler needs to be added in `<head>`:

...

```
<script src="https://unpkg.com/@babel/standalone@7/babel.min.js"></script>
```

...

- Attribute `type="text/babel"` needs to be added to the script tag containing JSX code.
- JSX can also be kept in an external script and referenced as follows:

...

```
<script type="text/babel" src="/App.jsx"></script>
```

...

More on JSX

Embedding expressions

```
const name = "Rahul Dravid";  
const element = <h1>Hello,{name}</h1>;
```

```
const element = (  
  <h1 className="greeting">  
    Hello, world!  
  </h1>  
) ;
```

Note: class HTML attribute needs to be className in JSX

Functions with expressions

```
function formatName(user) {  
  return user.firstName + ' ' +  
  user.lastName;  
}  
  
const user = {  
  firstName: "Rahul",  
  lastName: "Dravid"  
};  
  
const element = (  
  <h1>  
    Hello, {formatName(user)}!  
  </h1>  
) ;
```

A Simpler React Code with JSX

```
<!DOCTYPE HTML>
<html>
<head>
<title>React with JSX</title>
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
<script src="https://unpkg.com/@babel/standalone@7/babel.min.js"></script>
</head>
<body>
<div id="content"></div>
<script type="text/babel">
const element = (
  <div title="Outer div">
    <h1>Hello World!</h1>
  </div>);
ReactDOM.render(element, document.getElementById('content'));
</script>
</body>
</html>
```

JSX



A Ticking Clock

```
...  
...  
<script type="text/babel">  
  
  <div id="clock"></div>  
  
  function tick() {  
    const element = (  
      <div>  
        <h1>It is {new Date().toLocaleTimeString()}.</h1>  
      </div>  
    );  
    ReactDOM.render(element, document.getElementById('clock'));  
  }  
  setInterval(tick, 1000);  
</script>  
...  
...
```

Note: Inspect this code using browser tools to see how React updates only what's necessary! (i.e the text node here)

Dealing with Changes

```
...  
var destination=document.getElementById("content");  
ReactDOM.render(  
  <div>  
    <h3>Cricket</h3>  
    <h3>Football</h3>  
    <h3>Tennis</h3>  
    <h3>Baseball</h3>  
  </div>,  
  destination  
) ;  
...
```

What changes would you need to make to convert all the `<h3>` tags to `<h1>`?

Clearly the change needs to be duplicated for every instance of `<h3>`

Imagine the same changes but on a larger scale with greater complexity.

Some questions you need to ask:

How does one tackle such time consuming requests?

Which programming constructs bring in the concept of reusability?

React has a solution.....

React Classes and Components

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- A valid React component is one that accepts a single “props” (which stands for properties) object argument with data and returns a React element.
- These reusable pieces of code output HTML.
- A component can be created as Function Component or Class Component.
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.
- Typically, React classes (ES6 classes) are used to create real components.
- These classes can then be reused within other components, pass data between them, handle events, and so much more.

React Classes and Components

- React classes are created by extending `React.Component`, the base class from which all custom classes must be derived like so:

```
...  
class Customclass extends React.Component {  
  ...  
}  
...
```

- Methods with special meaning to React called the Lifecycle methods which provide hooks into various stages of the component formation and other events can be implemented within these classes.
- `render()` is one that must be present, otherwise the component will have no screen presence**
- The `render()` function is supposed to return an element (which can be either a native HTML element such as a or an instance of another React component)**

Creating a Simple Component

```
const element = (  
  <div title="Outer div">  
    <h1>Hello World!</h1>  
  </div>);  
ReactDOM.render(element,  
document.getElementById('conte  
nt'));
```



```
class HelloWorld extends React.Component  
{  
  render() {  
    return (  
      <div title="Outer div">  
        <h1>Hello World!</h1>  
      </div>  
    );  
  }  
}
```

The code on the left produces an output on the browser. Now that the component is created, how do we see it on the browser?

Rendering a Component

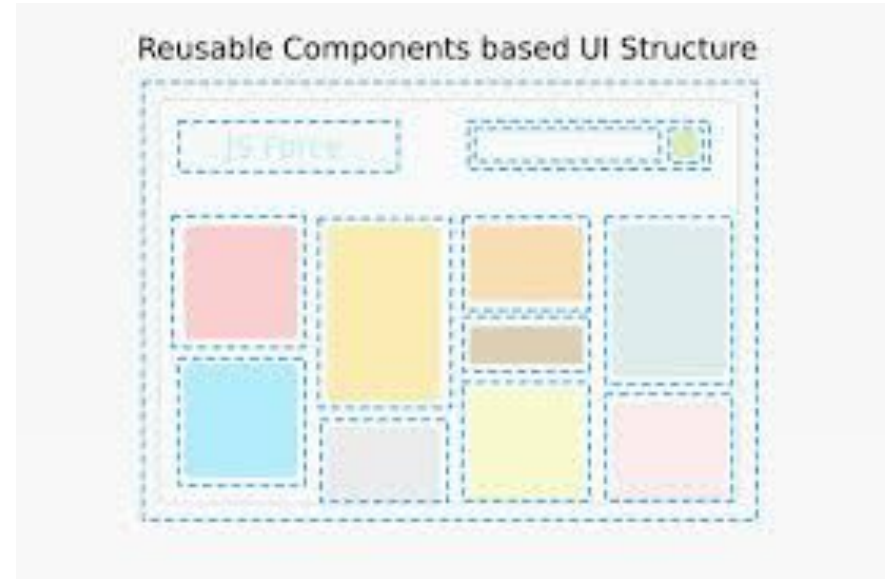
In the previous component, the render method is made to return the JSX that represents the **Hello world!** text. All that remains is to actually use this component. The way you use a component once you've defined it is by **calling** it using ReactDOM.render.

`const element = <HelloWorld />;` —————→ Creating an instance of HelloWorld Class

`ReactDOM.render(element, document.getElementById('content'));`

Composing Components

- So far we have seen how to build a component by putting together built-in React components that are HTML element equivalents.
- It's possible to build a component that uses other user-defined components as well.
- Component composition is one of the most powerful features of React.
- The UI can be split into smaller independent pieces so that each piece can be coded and reasoned in isolation, making it easier to build and understand a complex UI.
- It encourages reuse.



Composing Components

Here are a few things to remember when composing components:

- Larger components should be split into fine-grained components when there is a logical separation possible between the fine-grained components.
- When there is an opportunity for reuse, components can be built which take in different inputs from different callers.
- React's philosophy prefers component composition in preference to inheritance.
- In general, remember to keep coupling between components to a minimum. (coupling is where one component needs to know about the details of another component, including parameters or properties passed between them)

A Basic Composition

For example, we can create an IDcard component that renders a photo and a name:

```
...  
<div id="contents"></div>  
<script type="text/babel">  
  class Name extends React.Component {  
    render() {  
      return (<div>John</div>);  
    }  
  }  
  
  class Photo extends React.Component {  
    render() {  
      return (<div>Placeholder for John's  
Photo</div>);  
    }  
  }
```

```
class IDcard extends React.Component {  
  render() {  
    return(<div>  
      <Photo/>  
      <Name/>  
    </div>  
  );  
}  
  
ReactDOM.render(<IDcard/>,  
document.getElementById('contents'));  
</script>  
...
```

Styling Components

- Using plain old CSS is a perfectly viable way to style the content in your React-based apps
- React favors an inline approach for styling content that doesn't use CSS
- The way you specify styles inside your component is by defining an object whose content is the CSS properties and their values.
- Once you have that object, you assign that object to the JSX elements you wish to style by using the style attribute.
- This object has a specific convention containing a series of JavaScript key-value pairs.

Styling Components

- **Convention followed:**

- The keys are the same as the CSS style name, except that instead of dashes (like border-collapse), they are camel cased (like borderCollapse).
- The values are CSS style values, just as in CSS. Single word CSS properties (like padding, margin, color) remain unchanged.
- There is also a special shorthand for specifying pixel values; you can just use a number (like 4) instead of a string "4px".

Example: **const divStyle = {border: "1px solid silver", padding: 4};**

- Set the style attribute of the element that has to be styled to refer to that object.

Example: **<div style={divStyle}>**

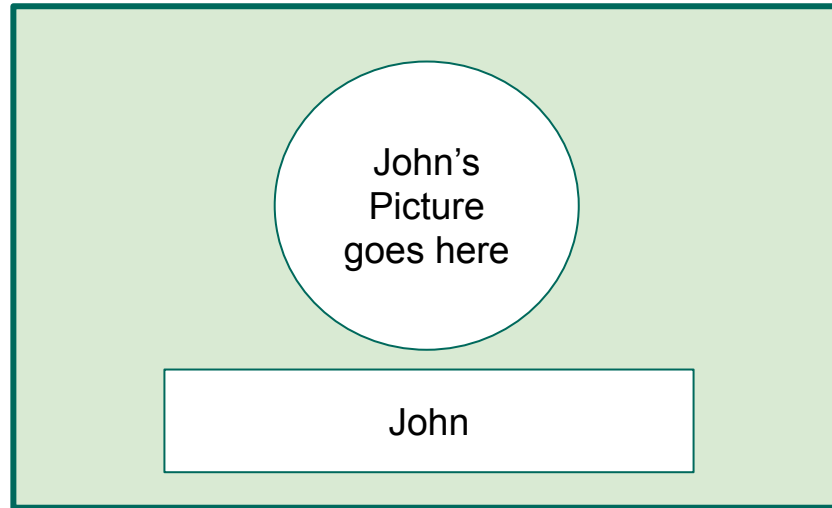
Component Composition with Styling

```
...  
<script type="text/babel">  
class Disciplines extends React.Component{  
  render() {  
    return <h2>This section details  
disciplines </h2>;  
  }  
}  
class Courses extends React.Component {  
  render() {  
    return <h2>This section details  
courses</h2>;  
  }  
}
```

```
class School extends React.Component{  
  render() {  
    const divStyle = {border: "5px solid  
black"};  
    return(  
      <div style={divStyle}>  
        Welcome to School!  
        <Disciplines/> <hr/>  
        <Courses/> <hr/>  
      </div>);  
    }  
  ReactDOM.render(<School/>,document.body) ;  
</script> ...
```

Try It Yourself: ID Card

Experiment with styling using the ID card component demonstrated earlier to create a static ID Card like so:



Feel free to add more styling!

Try It Yourself: Nested Components

Replicate the given output using React components. Styling is optional but recommended.



Try It Yourself: Issue Tracker

Create an Issue Tracker by composing components to produce the output given below

Pro MERN Stack +

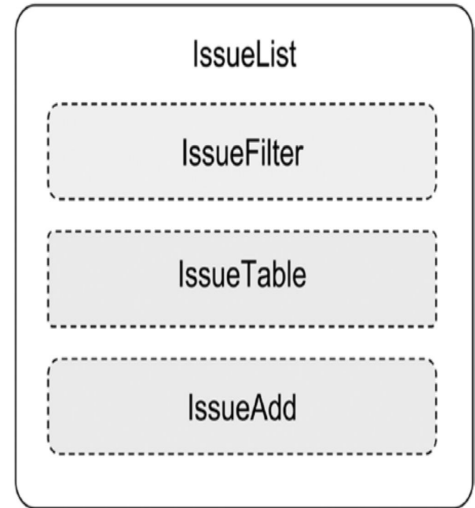
Issue Tracker

This is a placeholder for the issue filter.

This is a placeholder for a table of issues.

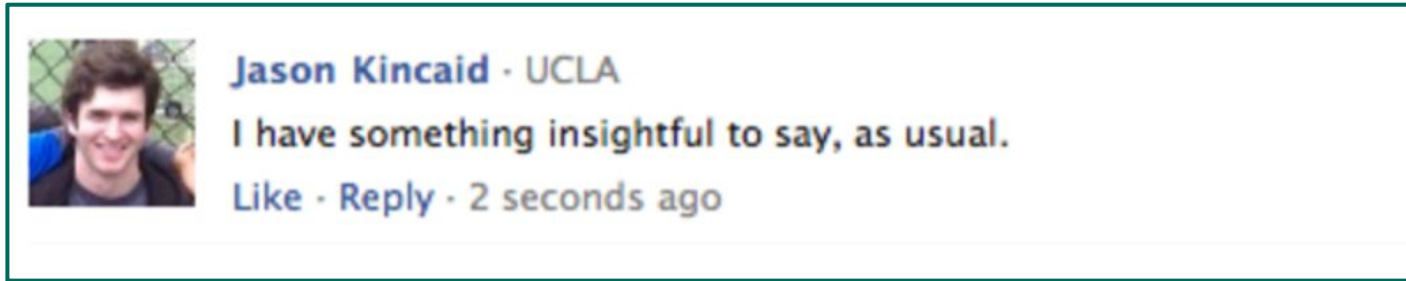
This is a placeholder for a form to add an issue.

Hint:

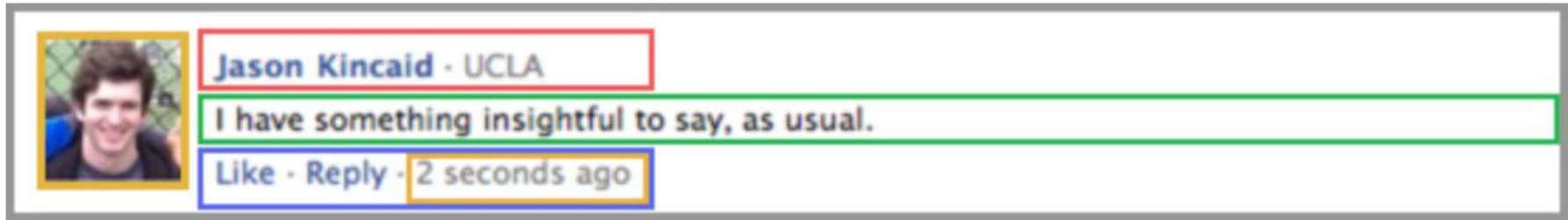


Try It Yourself: Facebook Comment Card

Identify the likely components and replicate the given output using React components. Styling is optional but recommended.



Try It Yourself: Facebook Comment Card



Passing Data Using Properties (props)

Composing components without any variables is not so interesting. It should be possible to pass different input data from a parent component to a child component and make it render differently on different instances.

- Props are arguments passed into React components.
- Props are received in React's class component via the 'this' instance of the class.
- Props enable you to pass variables from one to another component down the component tree.
- Props can be anything from strings, integers over objects to arrays
- There is **no way in React to set props**.
- A component must never modify its own props.

Using props

```
... <div id="contents"></div>
<script type="text/babel">
class Greeting extends React.Component {
  render() {
    return (<p>Welcome {this.props.name}</p>);
  }
}
const element = <div>
  <Greeting name="John"/>
  <Greeting name="Jane"/>
</div>;
ReactDOM.render(element, document.getElementById('contents'));
</script> ...
```

1. Read from this.props using the same attribute used while rendering.
2. When rendering the component, add the prop to the component using the attribute-like syntax who="Earth" as shown.

More on props

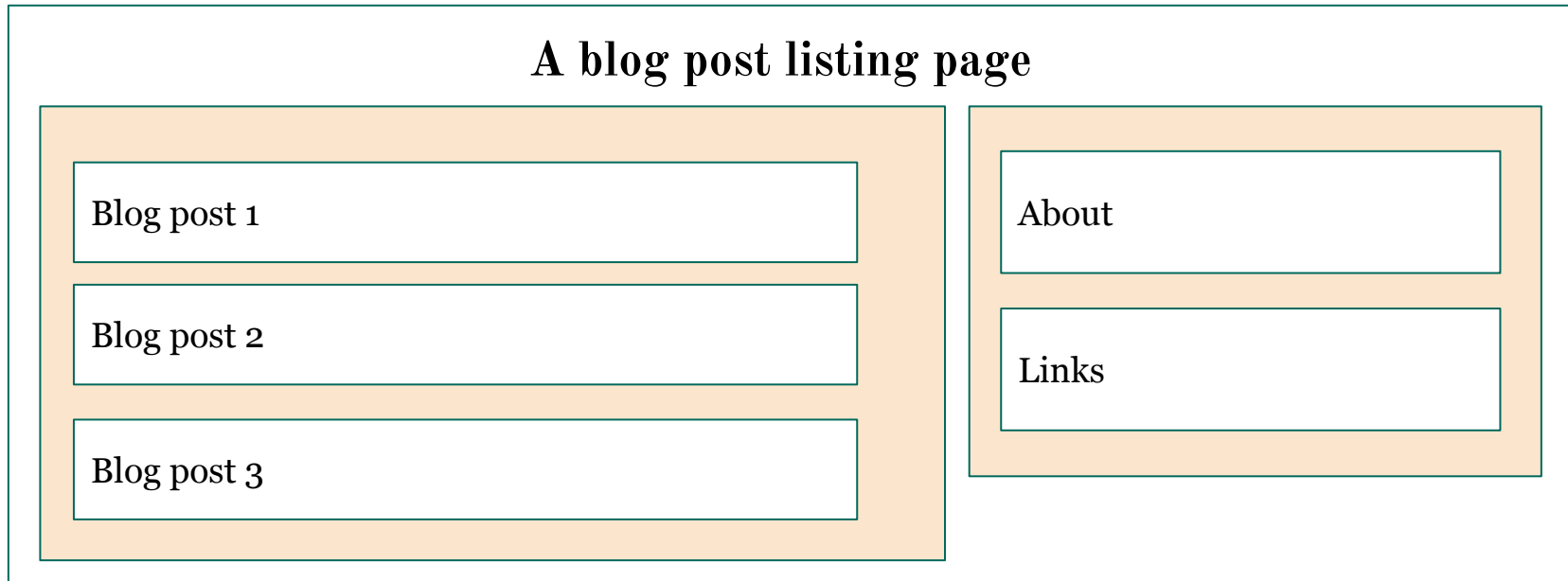
```
...<div id="contents"></div>
<script type="text/babel">
class Detail extends React.Component {
  render() {
    return <p>{this.props.message}</p>;
  }
}
ReactDOM.render(
  <Detail message="This is coming from props!"
/>,
  document.getElementById('contents')
);
</script> ...
```

If you created a component and gave it some props you can change those props later if you want to, but props you receive (i.e., `this.props`) should *not* be changed!

React has a one way data flow. Parent components pass props down to its children. Children components can not pass props up to their parent component.

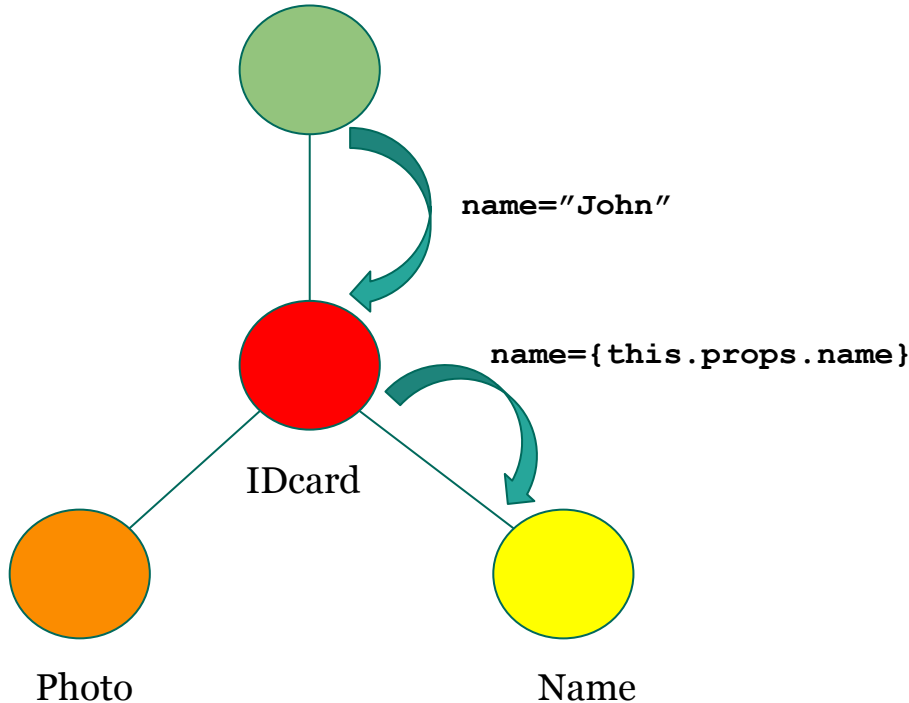
Try It Yourself: Blog Layout

Create a Blog layout by composing components to produce the output given below. Use CSS styling wherever required.



Passing Data from Parent Component to Child

ReactDOM.render()



Let us revisit the ID card component. It was a static abstraction for Photo and Name.

Now, in order to make it reusable we need to pass input from ID Card to its children. For now, we shall focus only on passing name.

Using props, we pass properties from one component to its children until the intended component receives it.

Revisiting IDcard

```
...  
<div id="contents"></div>  
<script type="text/babel">  
  class Name extends React.Component {  
    render() {  
      return (<p>{this.props.name}</p>) ;}  
    }  
  }  
  class Photo extends React.Component {  
    render() {  
      return (<p>Placeholder for John's  
Photo</p>) ;}  
    }  
  }  
}
```

```
class IDcard extends React.Component {  
  render() {  
    return (<div>  
      <Photo/>  
      <Name name={this.props.name}/>  
    </div>  
  ) ;}  
}  
ReactDOM.render(<IDcard name="John"/>,  
document.getElementById('contents')) ;  
</script>  
...
```

As an exercise, try taking name as input from the user using a textfield.

Try It Yourself: Revisiting Issue Tracker

Construct the IssueTable component, which is essentially a `<table>`, with a header row and two columns (ID and title), and two IssueRow components, content of which is passed from IssueRow component. Specify an inline style for the table to indicate a collapsed border and use the same rowStyle variable to specify the header row styles, to make it look uniform

Pro MERN Stack

Issue Tracker

This is a placeholder for the issue filter.

ID	Title
1	Error in console when clicking Add
2	Missing bottom border on panel

This is a placeholder for a form to add an issue.

Note:
Try to use
`this.props.children!`

State

- React has another special built-in object called state, which allows components to create and manage their own data.
- State is a plain JavaScript object used by React to represent an information about the component's current situation.
- The difference is while a “normal” variable “disappears” when their function exits, the state variables are preserved by React.
- The state essentially holds the data, something that can change, as opposed to the immutable properties in the form of props.
- This data is component specific and can change over time.

Initializing State

- The state of a component is captured in a variable called **this.state** in the component's class, which should be an object consisting of one or more key-value pairs, where each key is a state variable name and the value is the current value of that variable.
- **constructor()** runs before anything happens in a component and it is the only place where you should assign **this.state** directly
- To understand this concept better, let's build a component that consists of a clickable button. The objective of this component is to display the number of times said button is clicked.

Initializing State

```
...<div id="contents"></div>
<script type="text/babel">
  class ClickMe extends React.Component{
    constructor() {
      super();
      this.state={clicks: 0};
    }
    render() {
      return(
        <h1> Number of clicks: {this.state.clicks}
      </h1> );
    }
  }
  ReactDOM.render(<ClickMe/>,
    document.getElementById('contents'));
</script> ...
```

Click Me!

Number of clicks: 0

So far, this looks like your average static page.
Does anything happen when you click the button?
How do make it interactive?

State Changes

- State allows components to respond to user input and other events.
- Any changes that take place within the component class can be rendered using state.
- A change in the state happens based on user-input, triggering an event, and so on.
- React components (with state) are rendered based on the data in the state which holds the initial information.
- So when state changes, React gets informed and immediately re-renders the DOM – **not the whole DOM, but only the component with the updated state.**
- The **setState()** method triggers the re-rendering process for the updated parts.

Updating State

The state can only be assigned a value in the constructor. After that, the state can be modified, but only via a call to `React.Component's this.setState()` method. This method takes in one argument, which is an object containing all the changed state variables and their values.

Calls to `setState` are asynchronous - don't rely on `this.state` to reflect the new value immediately after calling `setState`

Example:

```
this.setState({  
  id: "2020"  
});
```

Updating State

```
class ClickMe extends React.Component{
  constructor() {
    super();
    this.state={clicks: 0};
    this.counter = this.counter.bind(this);
  }
  counter() {
    this.setState({clicks: this.state.clicks + 1});
  }
  render() {
    return (
      <div>
        <button onClick={this.counter}> Click Me! </button>
        <h1> Number of clicks: {this.state.clicks} </h1>
      </div>);
  }
}
```

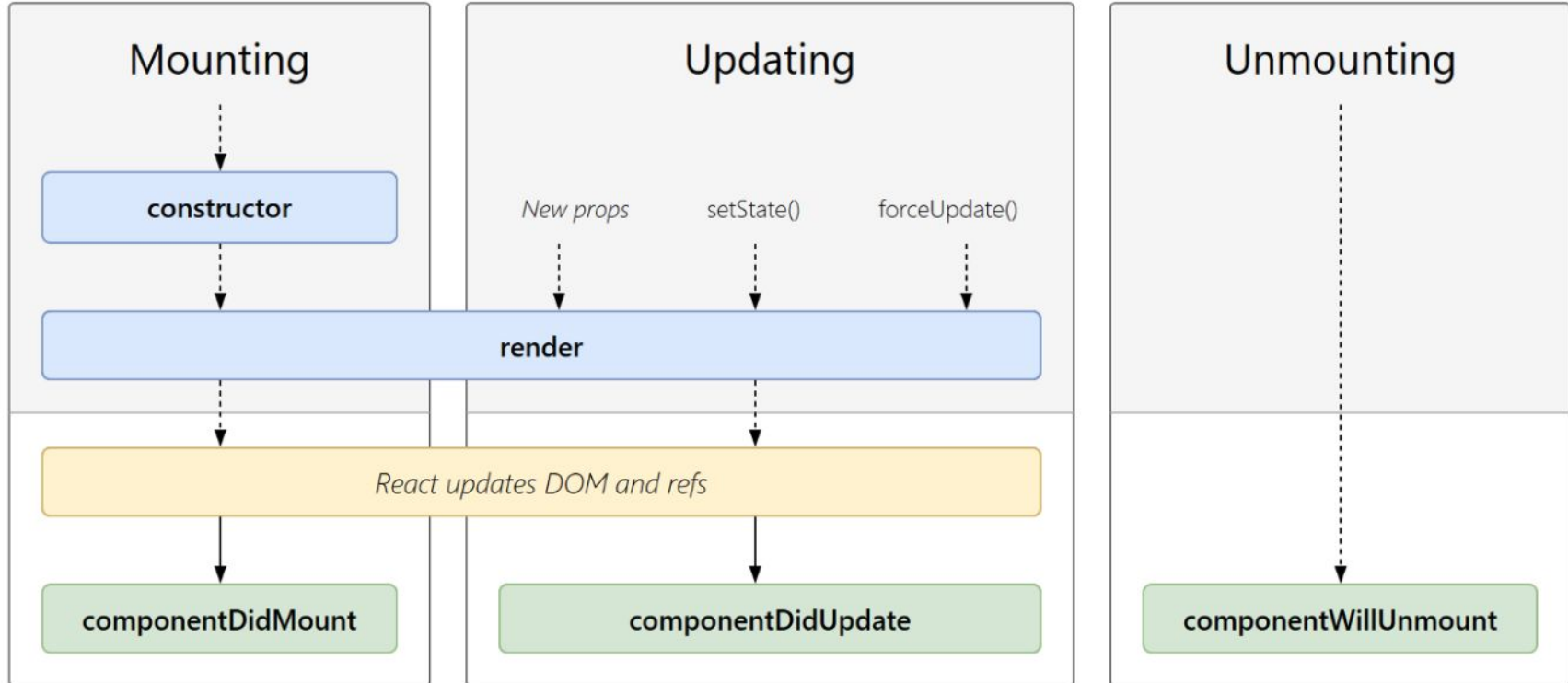

Props vs State

- Components receive data from outside with props, whereas they can create and manage their own data with state
- Props are used to pass data, whereas state is for managing data
- Data from props is read-only, and cannot be modified by a component that is receiving it from outside
- State data can be modified by its own component, but is private (cannot be accessed from outside)
- Props can only be passed from parent component to child (unidirectional flow)
- Modifying state should happen with the `setState ()` method.

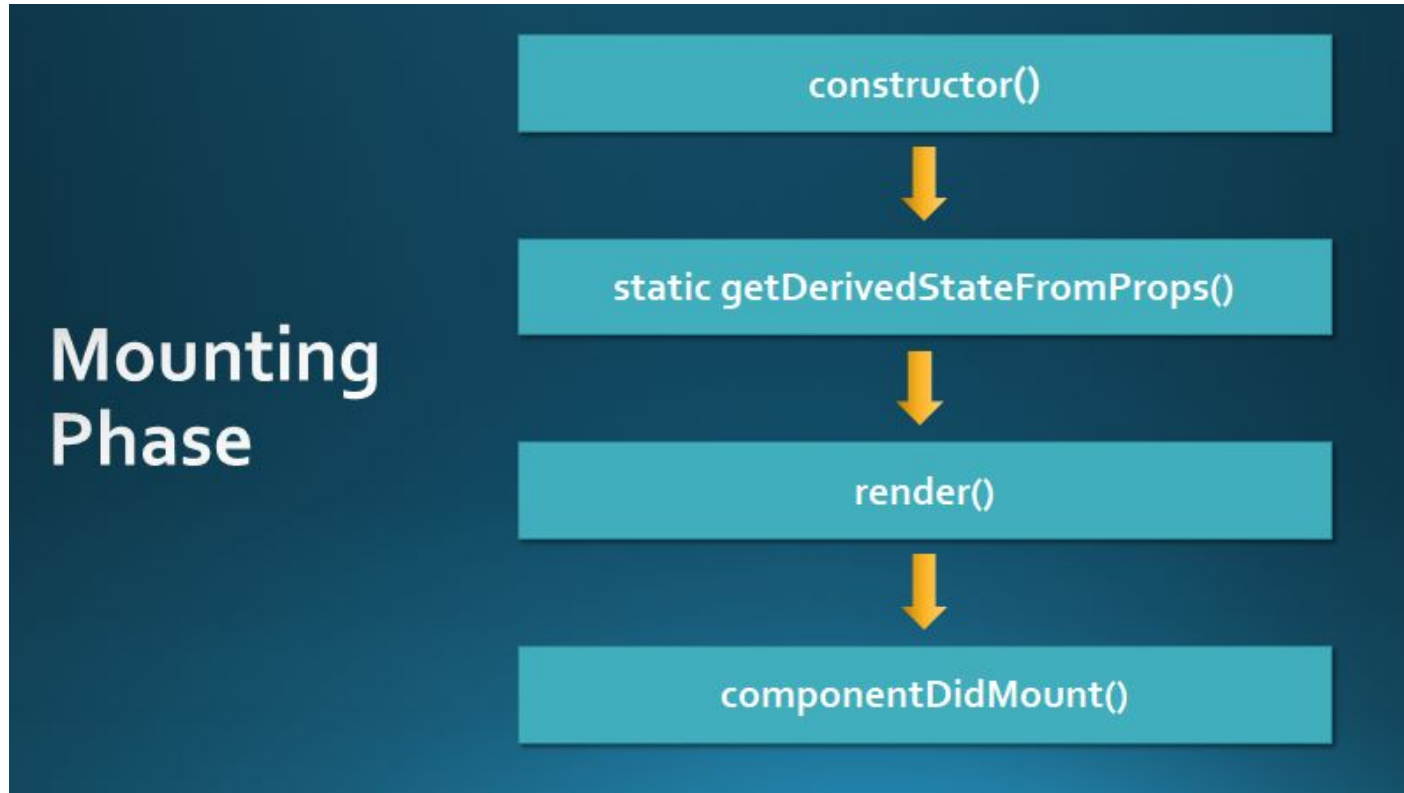
The Lifecycle of a React Component

- Each component goes through **three** phases: **mounting**, **updating** and **unmounting**.
- React components are created by being mounted onto the DOM, they change or grow through updates, and finally, they can be removed or unmounted from the DOM.
- These three milestones are referred to as the React **component lifecycle**.
- *A React Component **may or may not** go through all phases.*
- React provides specific lifecycle methods that can be used to perform specific actions in different phases. These are called React **component lifecycle methods**.

The Lifecycle of a React Component



LifeCycle: Mounting Phase



LifeCycle: Mounting Phase

This phase refers to the component's creation. This is where the component is added to the DOM.

Methods:

1. **constructor()**

- Called before anything else, when the component is initiated.
- Set up the initial **state** and other initial values here.
- **props** can be passed as arguments.
- **super(props)** is required before anything else which initiates the parent's constructor method and allows the component to inherit methods from its parent (**React.Component**)..

LifeCycle: Mounting Phase

2. `getDerivedStateFromProps()`

- Called right before rendering the element(s) in the DOM after `constructor()`
- Takes **state** as an argument, and returns an object with changes to the **state**

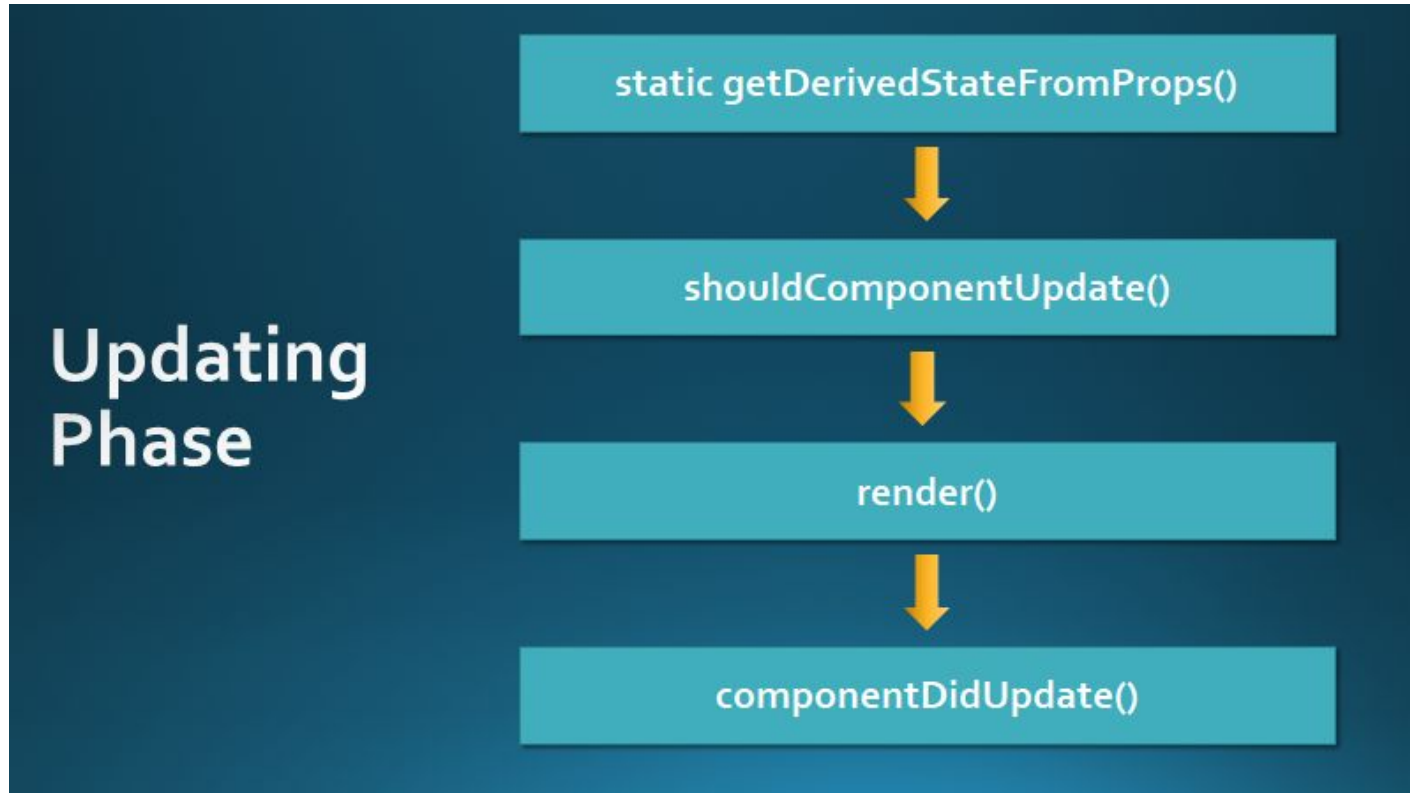
3. `render()`

- Outputs the HTML to the DOM.
- Required method

4. `componentDidMount()`

- Called after the component is rendered.
- Invoked after the render function is executed
- Calling `setState()` should be used here as it will call the render function again and handle asynchronous processes like fetch requests.

LifeCycle: Updating Phase



LifeCycle: Updating Phase

This second phase represents times where a component needs to update due to a change in its props or state. These changes can occur within the component or through the backend. These changes will trigger the render function again.

Methods:

1. `getDerivedStateFromProps()`

- First method that is called when a component gets updated.

2. `shouldComponentUpdate()`

- Next method to be invoked
- return a Boolean value that specifies whether React should continue with the rendering or not.
- Default: True

LifeCycle: Updating Phase

3. **render()**

- Also called when a component gets updated.
- Re-render the HTML to the DOM, with the new changes.
- If `shouldComponentUpdate()` returns true, the render function is invoked immediately.

4. **getSnapshotBeforeUpdate()**

- Access to the **props** and **state** *before* the update
- Use is uncommon

5. **componentDidUpdate()**

- called after the component is updated in the DOM.
- receives the former props and state values as arguments but it also receives the return value `getSnapshotBeforeUpdate()` as a third argument (if present).

LifeCycle: Unmounting Phase

Finally, the unmounting phase is where the component is removed from the DOM. This marks the end of a component's lifecycle. In this phase, we have one lifecycle method available to us.

Methods:

1. **componentWillUnmount()**

- Executed right before the component is unmounted from the DOM.
- Used to clean up anything that is needed to be removed before the component is destroyed.

Lifecycle Methods in Action: Timer

```
class Timer extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0  
    };  
    this.timerTick = this.timerTick.bind(this);  
  }  
  timerTick() {  
    this.setState((prevState) => {return {count: prevState.count + 1}});  
  }  
  componentDidMount() {  
    this.timer = setInterval(this.timerTick, 1000);  
  }  
  render() {  
    return (<h1>{this.state.count} seconds</h1> );  
  }  
}
```

Try It Yourself

Make use of the lifecycle methods to load the tracker and populate the issue table with the given data 2 seconds after the page loads:

{ID: 1, Status:"New", Owner:"John", created:"Fri Aug 8 2021", effort:5, duedate:"Sat Aug 9 2021", title:"Error in console when adding"}

Pro MERN Stack

Issue Tracker

This is a placeholder for the issue filter.

ID	Status	Owner	Created	Effort	Due Date	Title
----	--------	-------	---------	--------	----------	-------

This is a placeholder for a form to add an issue.

Try It Yourself

Create a button that changes the Status to “Completed” when clicked. The button can be placed in parent or child components. Adopt suitable techniques for updating the data.

Pro MERN Stack +

Issue Tracker

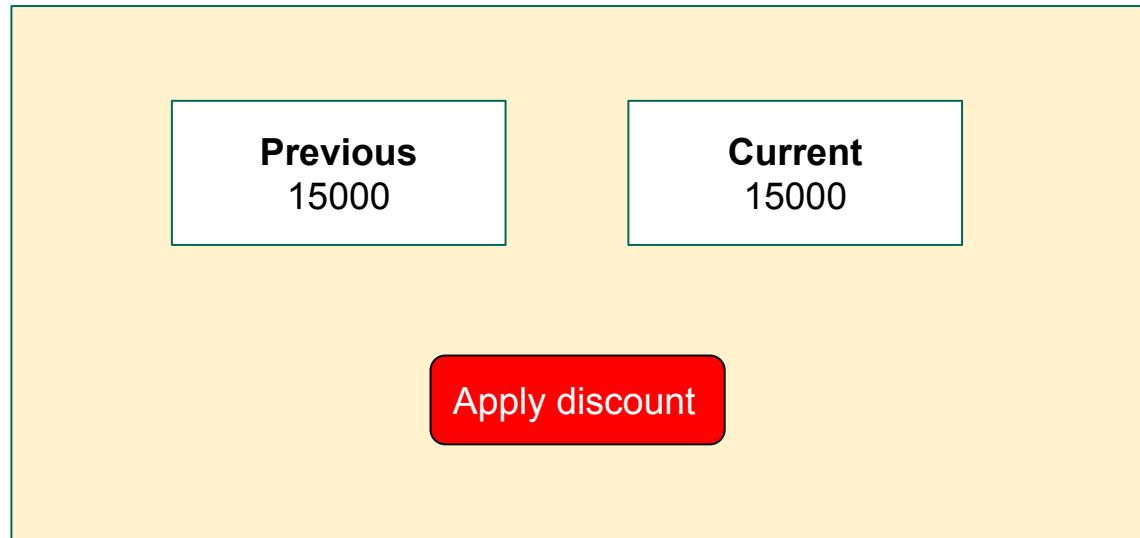
This is a placeholder for the issue filter.

ID	Status	Owner	Created	Effort	Due Date	Title
----	--------	-------	---------	--------	----------	-------

This is a placeholder for a form to add an issue.

Try It Yourself

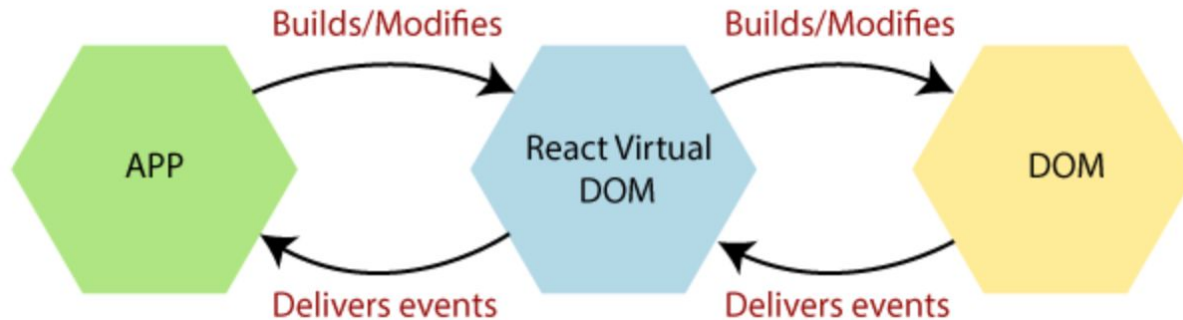
Create a web page that comprises of a button that applies a discount of 5% on 15000 when clicked. Previous prices should also be displayed. Make use of the lifecycle methods. Initially:



Event Handling in React

Most of the time, you don't have static UIs; you need to build elements that are smart enough to respond to user actions. So far in the examples we've seen, we brought in a certain level of user interaction such as performing some action on a button click.

By now, you would have realized that handling events with React elements is very similar to handling events on DOM elements:



However, React Event Handling comes with its own differences!

Firstly: lowercase vs camelCase

- React events are named using camelCase, rather than lowercase.

HTML



```
<button onclick="doSomething()">  
  Do something  
</button>
```

React



```
<button onClick={this.doSomething}>  
  Do something  
</button>
```


Secondly: Function vs String

- With JSX you pass a function as the event handler, rather than a string. Also `this.doSomething` wasn't invoked. Moreover, only one statement is made use of. This contrasts with HTML/JS where `onclick` can include any arbitrary amount of JavaScript code:

HTML

```
<button onclick="doSomething();  
console.log('Something was done!');  
doSomethingElse();">  
    Run multiple functions  
</button>
```

React

```
<button onClick={this.doSomething}>  
    Run one function  
</button>
```

Thirdly: return false vs preventDefault

- Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly.

HTML

```
<form onsubmit="console.log('You clicked  
submit.');" return false">  
  <button type="submit">Submit</button>  
</form>
```

React

```
function Form() {  
  function handleSubmit(e) {  
    e.preventDefault();  
    console.log('You clicked submit.');  }  
  return (<form onsubmit={handleSubmit}>  
    <button type="submit">Submit</button>  
    </form>);  
}
```

A List of Events

Here is a list of some familiar events that are supported by React:

Mouse

- `onClick`
- `onContextMenu`
- `onDoubleClick`
- `onMouseDown`
- `onMouseEnter`
- `onMouseLeave`
- `onMouseMove`
- `onMouseOut`
- `onMouseOver`
- `onMouseUp`

Image

- `onLoad`
- `onError`

Selection

- `onSelect`

Form

- `onChange`
- `onInput`
- `onSubmit`

Focus

- `onFocus`
- `onBlur`

Keyboard

- `onKeyDown`
- `onKeyPress`
- `onKeyUp`

UI

- `onScroll`

Binding Events

Event Binding tells the browser that a particular function should be called whenever a definite event occurs i.e

whenever an event is triggered, the function which is bound to that event should be called.

Essentially, event binding allows you to add an event handler for a specified event.

This can be used to bind to any event, such as keypress , mouseover or mouseout .

Since class methods are not bound by default, it's necessary to bind functions to the class instance so that the **this** keyword would not return “undefined”.

Binding in the constructor() method

Bind creates a new function that will have **this** set to the first parameter passed to bind().

```
class BindingEventMethod1 extends React.Component {  
  constructor() {  
    super();  
    this.callOnClick = this.callOnClick.bind(this);  
  }  
  callOnClick() {  
    console.log("Button is clicked");  
  }  
  render() {  
    return <button onClick={this.callOnClick}> Click me!</button>  
  }  
}
```

In the constructor, **this** refers to the class component.

Binding with Arrow Functions

You can handle events in class components by binding them with the fat arrow function.

```
class BindingEventMethod2 extends React.Component {  
  handleEvent = event => {  
    alert("Button was clicked");  
  };  
  
  render() {  
    return (  
      <button onClick={this.handleEvent}>Click me!</button>  
    );  
  }  
}
```

An arrow function does not have its own **this**; the **this** value of the enclosing execution context is used.

Passing Parameters

```
class ExampleComponent extends React.Component{
  sayHello(name) {
    alert("Hello " + name);
  }
  render()
  {
    return (
      //<button className="button" onClick={this.sayHello('James')}>Click for a
      greeting!</button>

      <button className="button" onClick={() => this.sayHello('James')}>Click
      for a greeting!</button>
    );
  }
}
```

If the fat arrow notation was not used, the function would execute on mount!

Accessing Event Data: SyntheticEvent

Whenever you call an event handler within ReactJS, they are passed an instance of SyntheticEvent. A SyntheticEvent event has all of its usual properties and methods. These include its type, target, mouse coordinates, and so on.

React defines these synthetic events according to the W3C spec to take care of cross-browser compatibility. We can use the event's data like this:

```
class Clicker extends React.Component {
  handleClick = (event) => {
    console.log(event); // prints the event object
    console.log(event.type); // prints 'click'
  };

  render() {
    return <button className="button" onClick={this.handleClick}>Click me!</button>;
  }
}
```


SyntheticEvent Properties: MouseEvent

A SyntheticEvent that wraps a MouseEvent will have access to mouse-specific properties such as the following:

`boolean altKey`

`number button`

`number buttons`

`number clientX`

`number clientY`

`boolean ctrlKey`

`boolean getModifierState(key)`

`boolean metaKey`

`number pageX`

`number pageY`

`DOMEventTarget relatedTarget`

`number screenX`

`number screenY`

`boolean shiftKey`

SyntheticEvent Properties: KeyboardEvent

A SyntheticEvent that wraps a KeyboardEvent will have access to keyboard-related properties such as the following:

<code>boolean altKey</code>	<code>string locale</code>
<code>number charCode</code>	<code>number location</code>
<code>boolean ctrlKey</code>	<code>boolean metaKey</code>
<code>boolean getModifierState(key)</code>	<code>boolean repeat</code>
<code>string key</code>	<code>boolean shiftKey</code>
<code>number keyCode</code>	<code>number which</code>

React documentation provides a greater look into all the types of supported events and their associated properties:

<https://reactjs.org/docs/events.html>

More on the Event Object

```
class KeyboardEvents extends React.Component{
    handleKeyPress(e) {
        alert( "You pressed a key with code: " + e.charCode )
    }
    render() {
        return (
            <div>
                <input type="text" placeholder="Enter a character"
className="input"onKeyPress={ (e) => this.handleKeyPress(e) } />
            </div>
        );
    }
}
```

More on the Event Object

```
class MouseEvents extends React.Component {
  state = {
    mouseX: 0,
    mouseY: 0
  }
  handleMouse = (e) => {
    this.setState({ mouseX: e.screenX, mouseY: e.screenY });
  }
  render() {
    return (
      <div className="div1">
        <div onMouseMove={this.handleMouse} >
          <p> X: {this.state.mouseX}px </p>
          <p> Y: {this.state.mouseY}px</p>
        </div>
      </div>);
  }
}
```

Try It Yourself: Get Current Date

Simulate the below to obtain the current date string on the click of a button.

Current Time:

Sat Oct 16 2021 17:10:00 GMT+0530 (India Standard Time)

Get Current Time!

Try It Yourself: Character Log

Simulate the below to display characters as and when it is typed into a text field



Type something here

Refs

Refs provide a way to access DOM nodes or React elements created in the render method.

It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements.

According to React.js documentation some of the best cases for using refs are:

- managing focus
- text selection
- media playback
- triggering animations
- integrating with third-party DOM libraries.

Usually props are the way for parent components to interact with their children. However, in some cases you might need to modify a child without re-rendering it with new props. That's exactly when refs attribute comes to use.

Creating Refs

- Refs are created using `React.createRef()`
- It can be assigned to React elements via the **ref** attribute.
- Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

**Although
tempting, don't
Overuse Refs**

```
class MyComponent extends React.Component
{
  constructor(props) {
    super(props);

    this.myRef = React.createRef();
  }

  render() {
    return <div ref={this.myRef} />;
  }
}
```


Accessing Refs

When a ref is passed to an element in **render**, a reference to the node becomes accessible at the **current** attribute of the ref.

```
const node = this.myRef.current;
```

The value of the ref differs depending on the type of the node:

- When used on a HTML element, the ref created in the constructor with `React.createRef()` receives the underlying DOM element as its current property.
- When used on a custom class component, the ref object receives the mounted instance of the component as its current.

Adding Ref to a DOM element

```
class CustomTextInput extends React.Component {  
  constructor() {  
    super();  
    this.textInput = React.createRef();  
    this.focusTextInput = this.focusTextInput.bind(this);  
  }  
  focusTextInput() {  
    this.textInput.current.focus();  
  }  
  render() {  
    return (  
      <div>  
        <input className="input" type="text" ref={this.textInput} />  
        <input className="button" type="button" value="Focus the text input"  
onClick={this.focusTextInput} />  
      </div>);  
  }  
}
```

Attaching to <input>
using ref

Adding Ref to a Class Component

```
class AutoFocusTextInput extends React.Component {  
  constructor() {  
    super();  
    this.textInput = React.createRef();  
  }  
  
  componentDidMount() {  
    this.textInput.current.focusTextInput();  
  }  
  
  render() {  
    return (  
      <CustomTextInput ref={this.textInput} />  
    );  
  }  
}
```

The `<input>` field
gets focussed on
mount!

Callback Refs

```
class CustomTextInput extends React.Component {
  constructor() {
    super();
    this.focusTextInput =
this.focusTextInput.bind(this);
  }

  focusTextInput() {
    this.textInput.focus();
  }

  render() {
    return ( <div>
      <input type="text" className="input"
ref={ (input) => { this.textInput = input } } />
      <input type="button" className="button"
value="Focus the text input"
onClick={this.focusTextInput} /></div> );
  }
}
```

React also supports another way to set refs called “callback refs”, which gives more fine-grain control over when refs are set and unset.

Instead of passing a ref attribute created by `createRef()`, you pass a function.

The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.

Rendering Using `.map()`

React really simplifies the rendering of lists inside the JSX by supporting the Javascript **`.map()`** method.

The **`.map()`** method in Javascript iterates through the parent array and calls a function on every element of that array. Then it creates a new array with transformed values.

It doesn't change the parent array.

```
const numbers = [1, 2, 3, 4, 5];  
  
const doubled = numbers.map((number) => number * 2);  
console.log(doubled); //[2, 4, 6, 8, 10]
```

Rendering Using .map()

Have the
developer tools
open when you
execute this
code!

```
class Student extends React.Component{  
  render() {  
    return (  
      <div>  
        <h3>  
          {this.props.name} is  
{this.props.age} years old  
        </h3>  
      </div>  
    );  
  }  
}
```

```
class List extends React.Component {  
  students = [  
    { id: 1, name: "Amal", age: 25 },  
    { id: 2, name: "Mark", age: 32 },  
    { id: 3, name: "Sithum", age: 28 },  
    { id: 4, name: "Tony", age: 30 }  
  ];  
  studentList = this.students.map(student  
=> (  
    <Student name={student.name}  
age={student.age} />  
  ));  
  render() {  
    return  
    <div>{this.studentList}</div>;  
  }  
}
```

Keys

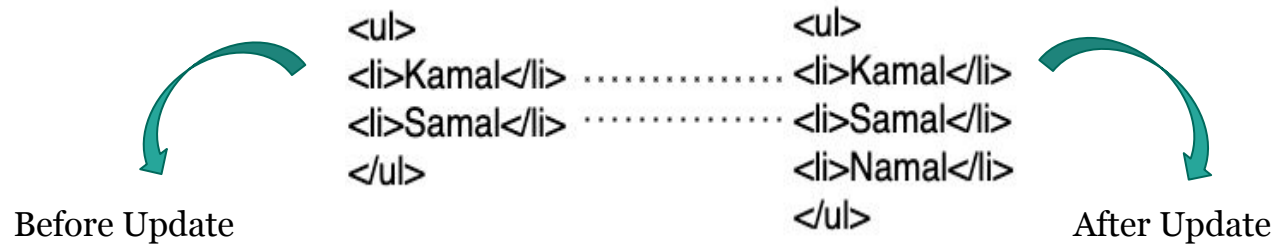
- A key is a unique identifier.
- Keys help React identify which items have changed, are added, or are removed.
- It is useful when we dynamically created components or when users alter the lists.
- The best way to pick a key is to choose a string that uniquely identifies the items in the list.
- Keys used within arrays should be unique among their siblings. However, they don't need to be globally unique.

```
const stringLists = [ 'One', 'Two',  
  'Three', 'Four', 'Five' ];  
  
const updatedLists = stringLists.map(  
  (strList) => {  
    <li key={strList.id}> {strList} </li>;  
  } );
```

The Importance of Keys

Keys are used to uniquely identify elements in a list and it helps React identify what is added, updated and changed. Apart from that, it helps in efficiently updating the DOM!

Consider the scenario of updating a list like so:



In the above case, React will verify that the first and second elements have not been changed and adds only the last element to the list.

The Importance of Keys

Now, consider a change in the updated list like so:

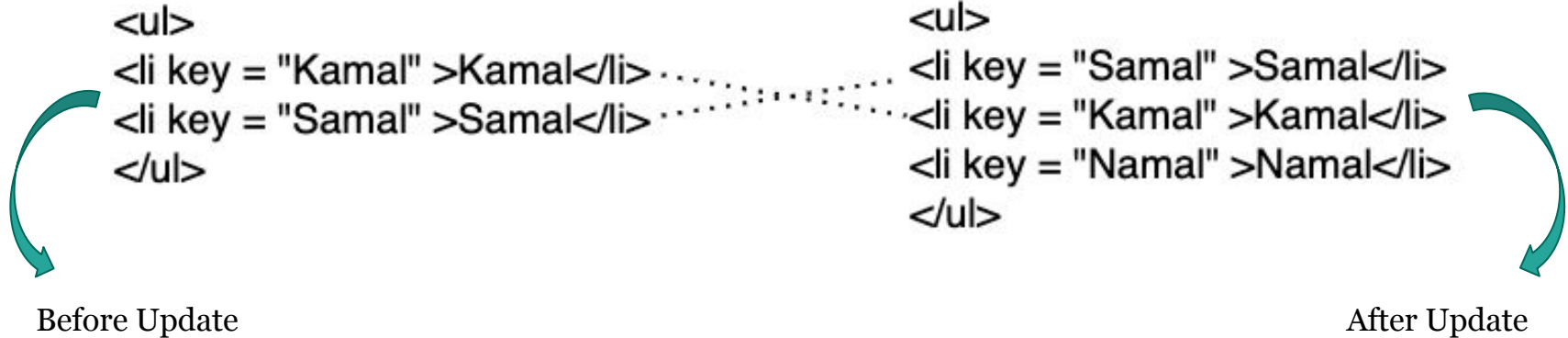


In this case, React cannot identify that “Kamal” and “Namal” have not been changed.

Therefore, it will update three elements instead of one which will lead to a waste of performance.
To solve this, keys are used.

The Importance of Keys

Using keys, React can identify elements and figure out which to add, remove or update thereby enhancing the performance.



Using keys

```
class Student extends React.Component{  
  render() {  
    return (  
      <div>  
        <h3>  
          {this.props.name} is  
{this.props.age} years old  
        </h3>  
      </div>  
    );  
  }  
}
```

```
class List extends React.Component {  
  students = [  
    { id: 1, name: "Amal", age: 25 },  
    { id: 2, name: "Mark", age: 32 },  
    { id: 3, name: "Sithum", age: 28 },  
    { id: 4, name: "Tony", age: 30 }  
  ];  
  studentList = this.students.map(student  
=> (  
    <Student key={student.id}  
      name={student.name}  
      age={student.age} /> ));  
  render() {  
    return  
    <div>{this.studentList}</div>;  
  }  
}
```

Stateful Components

So far, we have built many components with the help of **state** for storing information about the component's state change in memory. Such components which utilize state are called stateful components.

A **stateful** component is always a *class* component.

A stateful component is dependent on its state object and can change its own state.

The component re-renders based on changes to its state, and may pass down properties of its state to child components as properties on a props object.

Stateful Components

When would you use a stateful component?

- When building element that accepts user input
- ..or element that is interactive on page
- When dependent on state for rendering, such as, fetching data before rendering
- When dependent on any data that cannot be passed down as props

A Stateful Component

```
class StateAndProps extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state={  
      value: '50'  
    }  
  }  
  render() {  
    return (  
      <div>  
        <p>{this.state.value}</p>  
        <p>{this.props.value}</p>  
      </div>  
    )  
  }  
}
```

This stateful
component uses
props too!

Stateless Components

On the other hand, there also exist components which do not use **state**. Such components just print out what is given to them via props, or they just render the same thing.

For performance reasons and for clarity of code, it is recommended that such components (those that have only `render()`) are written as functions rather than classes: a function that takes in props and just renders based on it.

It's as if the component's view is a pure function of its props, and it is stateless. The `render()` function itself can be the component.

If a component does not depend on props, it can be written as a simple function whose name is the component name

Stateless Components

Stateless components are those components which don't have any state at all, which means you can't use *this.setState* inside these components.

It has no lifecycle, so it is not possible to use lifecycle methods such as *componentDidMount* and other hooks.

When react renders our stateless component, all that it needs to do is just call the stateless component and pass down the props.

A functional component is always a stateless component, but the class component can be stateless or stateful.

In stateless components, the props are displayed like *{props.name}*

There is no need for *'this'* keyword that has always been a significant cause of confusion and they are much easier to test.

Stateless Components

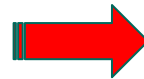
A **stateless** component is usually associated with how a concept is presented to the user.

When would you use a stateless component??

- When you just need to present the props
- When you don't need a state, or any internal variables
- When creating element does not need to be interactive
- When you want reusable code

Creating Stateless Components

```
class HelloWorld extends React.Component
{
  render() {
    return (
      <div title="Outer div">
        <h1>Hello World!</h1>
      </div>
    );
  }
}
```



```
function HelloWorld() {
  return (
    <div title="Outer div">
      <h1>Hello World!</h1>
    </div>
  );
}
```

Creating Stateless Components: Using props

```
...  
function HelloWorld(props) {  
  return (  
    <div title="Outer div">  
  
    <h1>{props.message}</h1>  
  
    </div>  
  );  
}  
  
ReactDOM.render(<div><HelloWorld message="Welcome"/></div>, document.body);  
...
```

Creating Stateless Components: Using props

```
...  
function App() {  
    const greeting = 'Hello Function Component!';  
    return <Headline value={greeting} />;  
}  
function Headline(props) {  
    return <h1>{props.value}</h1>;  
}  
ReactDOM.render(<App/>, document.body);  
...
```

Stateful vs Stateless

Stateful Components	Stateless Components
Also known as container or smart components.	Also known as presentational or dumb components.
Have a state	Do not have a state
Can render both props and state	Can render only props
Props and state are rendered like <code>{this.props.name}</code> and <code>{this.state.name}</code> respectively.	Props are displayed like <code>{props.name}</code>
A stateful component is always a <i>class</i> component.	A Stateless component can be either a functional or class component.

Stateful vs Stateless

```
class Store extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { sell: 'anything' };  
  }  
  render() {  
    return <h1>I'm selling {this.state.sell}</h1>;  
  }  
}
```

The **Store** component is stateful and the **Week** component is stateless.

```
class Week extends React.Component {  
  render() {  
    return <h1>Today is {this.props.day}</h1>;  
  }  
}
```

React Forms

Forms in React are handled primarily based on the manner in which data is managed. Before we move onto forms in React, let's understand how HTML typically handles form input. HTML form elements naturally keep some internal state. Form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. The DOM becomes responsible for handling the form data.

```
<form>
  <label>
    Name :
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

The data is handled
by DOM and
submitted using
default form
behavior

React Forms

There are two ways in which forms can be created in React:

- **Uncontrolled Components:** These React Components are traditional HTML form inputs which remember what you typed. Refs are used get the form values.
- **Controlled Components:** These are React components that render a form and also control what happens in that form on subsequent user input. This means that, as form value changes, the component that renders the form saves the value in its state. The controlled component is a way that you can handle the form input value using the state and to change the input value there is only one way to change it is using `setState`

Uncontrolled Components

```
class UncontrolledForm extends React.Component {  
  constructor() {  
    super();  
    this.handleSubmit = this.handleSubmit.bind(this);  
    this.input = React.createRef();  
  }  
  handleSubmit(event) {  
    alert('A name was submitted: ' + this.input.current.value);  
    event.preventDefault();  
  }  
  render() {  
    return (  
      <form onSubmit={this.handleSubmit}>  
        Name:<input className="input" type="text" ref={this.input} />  
        <input type="submit" value="Submit" />  
      </form>  
    );  
  }  
}
```

These forms rely on the DOM to remember the form inputs

Controlled Components

```
class ControlledForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }
  render() {
    return ( <form onSubmit={this.handleSubmit}><label>Name:
      <input type="text" value={this.state.value} onChange={this.handleChange} />
      </label> <input type="submit" value="Submit" />
    </form>); }
}
```

Controlled form components are defined with a value property.

Controlled Components

The value attribute allows us to handle `<input type="text">`, `<textarea>`, and `<select>` similarly

```
<textarea value={this.state.value}  
onChange={this.handleChange} />
```

```
<select>  
  <option  
value="grapefruit">Grapefruit</option>  
  <option value="lime">Lime</option>  
  <option selected  
value="coconut">Coconut</option>  
  <option value="mango">Mango</option>  
</select>
```

They all accept a value attribute that you can use to implement a controlled component.

Handling Multiple Inputs

```
class CustomForm extends React.Component {
  constructor() {
    super();
    this.state = {name:'', email: ''};
    this.handleChange=this.handleChange.bind(this)
  }

  handleChange(event) {
    this.setState({[event.target.name]: event.target.value});
  }

  render() { return (
    <div><form>
      <input type="text" name="name" value={this.state.name}
onChange={this.handleChange} placeholder="Name"/><br/>
      <input type="text" name="email" value={this.state.email}
onChange={this.handleChange} placeholder="Email" />
    </form> </div> ) }
}
```

To work with multiple input elements, add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name

Form Validation

Form validations on React can be performed using libraries like Redux Form, Formik.

Another approach would be to use built-in HTML5 form validation as it is declarative and easy to use.

- You can use required for all input

```
<input type="text" id="username" name="username" required/>
```

- You can use pattern attribute to enforce a certain pattern.

```
<input type="text" id="country_code" name="country_code" pattern="[A-Za-z]{3}">
```

- You can also use maxlength attribute

```
<input type="text" id="username" name="username" maxlength="10">
```

Constraint Validation API for Form Validation

The Constraint Validation API enables checking values that users have entered into form controls, before submitting the values to the server.

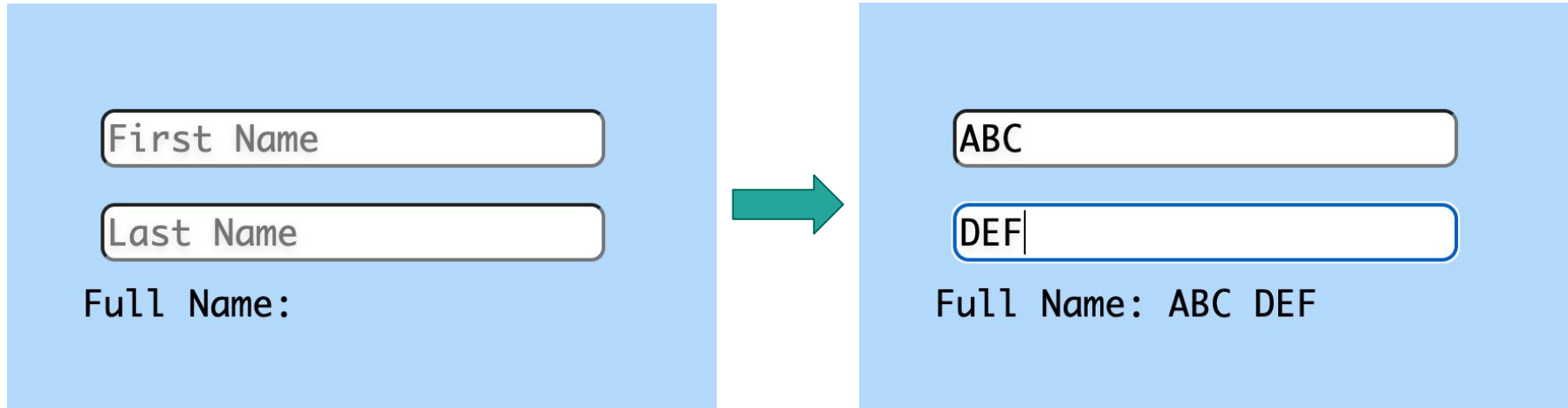
```
class ReactForms extends React.Component {  
  constructor() {  
    super();  
    this.form = React.createRef();  
    this.validate = this.validate.bind(this);  
  }  
  validate() {  
    this.form.current.reportValidity();  
  }  
  render() {  
    return (  
      <div> <form ref={this.form} onSubmit={e => e.preventDefault()}>  
        <input className="input" placeholder="required" required />  
        <input className="input" placeholder="Enter three digits" required pattern="\d{3}" />  
        <button className="button" onClick={this.validate}>Validate me</button></form></div>  
    );  
  }  
}
```

HTMLFormElement.reportValidity()
method returns true if the element's child controls satisfy their validation constraints.

Try It Yourself: A Simple Form

With the help of a form, accept first and last names from the user separately and display full name. Try to use one handler function for both the inputs.

Additionally, display an error message on entering numerical characters.



The diagram illustrates the state of a web form at two different points in time, connected by a green arrow pointing from left to right.

Initial State (Left):

- Two input fields are present: "First Name" and "Last Name".
- Below the fields is the label "Full Name:".

Final State (Right):

- The "First Name" field now contains the text "ABC".
- The "Last Name" field now contains the text "DEF".
- The "Full Name:" label is now followed by the output "ABC DEF".



THANK YOU

Prof. Spurthi N Anjan

Department of Computer Science and Engineering

spurthianjan@pes.edu

Sinduja Mullangi

Teaching Assistant

sinduja.mullangi@gmail.com